Abstract:

A parsing method is described based upon Knuth's LR(k) method of analysis. It provides greater generality and earlier detection of syntax errors than other formal methods used in compilers. Many space and time optimisations are available some of which are examined in detail. Time and space efficiency is evaluated using the Stanford University AlgolW compiler as a base for comparison since the parsing routine in this compiler is among the most efficient presently used.

Efficient LR(1) parsers

by

T. Anderson,
J. Eve
and
J. J. Horning.

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw Number 24 September, 1971.

© 1969 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
Erratum

Insert the following on page 20 immediately preceding the heading of section 4.2.

"Step 2b) must be modified to

b) a production number, \( p \) say; if \( j \in V_T \) then \( n_p \)
elements are popped from the stack and \( j \) is stored in a temporary location \( TEMP \). If
\( j \in V_N \) then \( n_p - 1 \) elements are popped from the stack. \( k \) is the state number now at the top
of the stack. \( 1 \) is \( A_p \), the LHS of the production \( p \).
ANDERSON, Thomas
Efficient LR(1) parsers. [By]
Newcastle upon Tyne: University of
Newcastle upon Tyne, Computing Laboratory
1971.
(University of Newcastle upon Tyne, Computing
Laboratory, Technical Report Series, no. 24)

12"

Added entries
EVE, James
HORNING, James Jay
UNIVERSITY OF NEWCASTLE UPON TYNE. Computing

Suggested classmarks (primary classmark underlined)
Library of Congress:
Dewey (17th):
U.D.C.:
510.7834
681.322.06 ; 681.326.3

Suggested keywords
COMPILERS
FORMAL LANGUAGES
LR(k) GRAMMARS
SYNTAX ANALYSIS

Abstract
A parsing method is described based upon Knuth's LR(k)
method of analysis. It provides greater generality
and earlier detection of syntax errors than other
formal methods used in compilers. Many space and
time optimisations are available some of which are
examined in detail. Time and space efficiency is
evaluated using the Stanford University Algo1W compiler
as a base for comparison since the parsing routine in
this compiler is among the most efficient presently used.

About the Author
Mr. Anderson is a postgraduate research student in the
Computing Laboratory of the University of Newcastle upon Tyne.
Dr. Eve is a senior lecturer in the Computing Laboratory of
the University of Newcastle upon Tyne.
Professor Horning is an assistant professor in the University of
Toronto.
Figure 1

Figure 2
1. Introduction

The widespread use of context free grammars to model the syntax of programming languages has resulted in the parsing problem for such languages receiving a great deal of attention both theoretically and practically.

It is known that, for parsing an arbitrary context free language, an amount of time proportional to the cube and an amount of space proportional to the square of the length of the input string is sufficient (see, for example, Earley 1970). For various subclasses of context free grammars these bounds can be considerably improved. Programming languages are generally amenable to analysis by parsing methods with time and space requirements which are linearly related to the length of the input string. Several such linear parsing algorithms are described in the survey article of Feldman and Gries (1968).

Historically, these parsers were classified into top down and bottom up categories according to the primary strategy adopted. In the former, starting from the principal nonterminal of a grammar, an attempt is made to produce a derivation of the input string. Bottom up methods start with the input string and try to produce a parse directly.

It was pointed out by Knuth (1965) that the bottom up strategy results in a parse which has the property, when viewed (in reverse) as a derivation, that the rightmost nonterminal in each sentential form is replaced in each step. The top down strategy results in a derivation in which the leftmost nonterminal in each sentential form is replaced at each step (Levis and Stearns, 1968); i.e. the bottom up strategy leads to a right canonical parse while the top down strategy leads to the left canonical derivation. It is probably the case that these are the appropriate formal notions embodied in the intuitive ideas of bottom up and top down. In his paper, Knuth defined a subclass of the context free languages, the LR(k) languages, which are precisely the set of languages for which the right canonical parse can be obtained deterministically (i.e. without backing up and changing any parsing decision already made) in a single pass over the input string from left to right, looking at most k symbols ahead on the input string to determine each parsing decision. Similarly Levis and Stearns consider the LL(k) languages, in which the left canonical derivation can be
obtained deterministically under the same constraints.

Both classes of language can be parsed in time and space linearly proportional to the length of the input string, but the fact that the \( \text{LL}(k) \) languages are properly contained in the \( \text{LR}(k) \) languages implies that the latter are potentially of greater practical interest. (It is true, however, that in the former category at any point in a parse only the first \( k \) terminal symbols generated by a production need to be matched with the input string to determine whether that production was used in the derivation. In the case of \( \text{LR}(k) \) languages it is not possible, in general, to determine whether a production has been used until \( k \) terminal symbols beyond the last generated by that production have been inspected).

The \( \text{LR}(k) \) property is associated with a grammar; a language is \( \text{LR}(k) \) by virtue of there existing at least one such grammar which generates it. Knuth has given two procedures which decide, when given a particular value of \( k \), whether a grammar is \( \text{LR}(k) \). The second of these can be modified to provide state tables. In the event that the grammar is \( \text{LR}(k) \), a relatively simple interpreter for scanning these tables, in conjunction with a stack, provides a parsing routine for the language. Knuth has also shown that any language which is \( \text{LR}(k) \) is \( \text{LR}(1) \). (If sentences of the language terminate with a unique symbol which does not appear elsewhere in a sentence, this result can be sharpened to \( \text{LR}(0) \)). Unfortunately, exploitation of this result in compilers founders on the fact that it involves generating an \( \text{LR}(1) \) or \( \text{LR}(0) \) grammar in which the phrase structure of the original grammar is corrupted.

In the case of programming languages, the need for more than one character of look ahead arises primarily at the lexical level due to the paucity of symbols available on many current input devices. The occurrence of ':' and ':= ' in Algol 60 is a well-known example. Typically, for efficiency if for no other reason, lexical analysis is separated from syntactic analysis, and the lexical scan can deal with look ahead problems at this level of analysis as a minor addition to its other tasks. It is also the case that construction of other than trivial programming language grammars is an extremely error prone process almost invariably resulting in grammars which are syntactically ambiguous. Assuming that lexical problems have
been treated separately, it appears that programming language
grammars largely meet the constraints which allow the use of the
LR(1) algorithm; changes to a grammar to accommodate deviations
from the constraints are slight. It is claimed that amending a
grammar to enable the use of the more restrictive SLR(1) algorithm
advocated in this paper is a small additional burden which can be
treated in conjunction with the problem of eliminating genuine
ambiguous.

The generality of the LR(k) algorithm, even for k=1, precludes
its use directly as a parser in a compiler or compiler writing system;
e.g. a program implementing the LR(1) algorithm was applied to an
Algol 60 grammar: the run was terminated with the insertion of the
100000th state-input entry into the state table, at which point 1237
states had been created. (See also Korenjak, 1969). The work
reported here addresses the problems raised by Knuth, of developing
algorithms that accept LR(k) grammars or special classes of them and
of mechanically producing efficient parsing programs.

2. **Definitions and Notation**

Let Ø denote the empty set and Λ denote the null string.

If X and Y are sets of strings and xy denotes the concatenation
of x and y, then XY = \{xy \mid x \in X \text{ and } y \in Y\}

If X = {Λ} then sets X^i are defined by

\[ X^{i+1} = X^i X \text{ for } i \geq 0 \]

and X* is defined by

\[ X^* = \bigcup_{i=0}^{\infty} X^i \]

A context free grammar (with an end marker) is a quadruple

\[ G = (V_N, V_T, P, S) \]

where

- \( V_N \) is a finite set of nonterminal symbols,
- \( V_T \) is a finite set of terminal symbols,
- \( V_N \cap V_T = \emptyset \) and V denotes \( V_N \cup V_T \)
- \( S \in V_N \) is the **start symbol** or principal nonterminal of the grammar
- \( \varepsilon \in V \) is the **endmarker**,
- \( S \) is the **start string**,
- \( P \) is a finite subset of \( V_N \times V^* \) called the set of productions.

Productions may be denoted \( A \rightarrow \alpha \), where \( A \in V_N, \alpha \in V^* \). A and \( \alpha \) are
the **left hand side** (LHS) and **right hand side** (RHS) of the production

\[ A \rightarrow \alpha \]

5.
It will prove convenient to number the productions (in some arbitrary manner) by the integers 1, 2, ..., s and to use the notation

\[ A_\rho \rightarrow X_{p_1}X_{p_2} \cdots X_{p_\rho}, \ n_\rho \geq 0 \]

to denote the \( p \)th member of the set of productions.

Unless otherwise stated

- \( A, B, C, \ldots \) denote members of \( V_N \)
- \( a, b, c, \ldots \) denote members of \( V_T \)
- \( X, Y \) denote members of \( V \)
- \( \alpha, \beta, \gamma, \ldots \) denote members of \( V^* \)

If \( G=\langle V_N, V_T, P, S, L \rangle \) is a context free grammar, then \( \alpha \) directly derives \( \beta \), (written \( \alpha \Rightarrow \beta \)) if there exist \( \gamma, \delta, \omega \in V^* \) and \( A \in V_N \) such that

\[ \alpha = \gamma A \delta, \ \beta = \gamma \omega \delta \text{ and } A \rightarrow \omega \text{ is in } P. \]

Also \( \alpha \) derives \( \beta \) (written \( \alpha \Rightarrow^* \beta \)) if there exist \( \omega_1, \omega_2, \ldots, \omega_n \in V^*_N \), \( n \geq 0 \) such that

\[ \alpha = \omega_1 \Rightarrow \omega_2 \Rightarrow \cdots \Rightarrow \omega_n = \beta. \]

If \( \omega_1, \omega_2, \ldots, \omega_n \in V^*_N \) then the derivation is a chain derivation.

The \( p \)th production is a chain production if \( n_\rho = 1 \) and \( X_{p_\rho} \in V_N \).

The \( p \)th production is left recursive if \( A_\rho = X_{p_1} \) and then \( A_\rho \)

is said to be directly left recursive.

A nonterminal symbol \( A \) is left recursive if \( A = A \rightarrow \omega \in V^* \).

If \( A \in V_N \) and \( \alpha \in V^* - A \) then we define the sets

- first \( (\alpha) = \{ X \in V | \alpha \Rightarrow X \omega \} \)
- follow \( (A) = \{ X \in V \cup S | \gamma \Rightarrow A X \} \) where \( \gamma, \omega \in V^* \)

3. The Constructor Algorithms

In this section algorithms are given for the construction of recognisers in the form of state tables. The fundamental algorithm is that of Knuth and the ensuing discussion presumes familiarity with it; however, the descriptions of the modified algorithms derived from it and presented here are self contained.

In Knuth's algorithm, a state \( S \) of a parse is defined as a set of triples of the form \([p, j, \omega]\). The appearance of \([p, j, \omega]\) in a state corresponds to the informal notion that in this state the parse has reached a point consistent with the propositions that

(a) production number \( p \) may have been used to generate the portion of the sentence currently being scanned,

* Knuth uses state set in place of state

6.
if production \( p \) has been used, the symbols \( X_{p1}, X_{p2}, \ldots, X_{pj} \) (or derivations thereof) of production \( p \) have already been recognised,

(c) if production \( p \) has been used, the \( k \) letter terminal symbol string \( \alpha \) may legitimately immediately succeed the current use of production \( p \).

It is appropriate to consider the reason for the large number of states arising in the LR(1) algorithm. Consider an LR(1) grammar in which a nonterminal symbol \( A \) occurs in the RHS of some production where it is immediately followed by a nonterminal \( B \); if \( p \) is the (index) number of a production defining \( A \), the LR(1) algorithm will generate a state containing \([ [p_1; b_1], [p_1; b_2], \ldots, [p_1; b_n] \] where \( b_i (i=1, \ldots, r) \) are members of \( V_T \) such that \( B^+ b_i \alpha \). (We assume for simplicity that neither \( A \) nor \( B \) generate the empty string).

Similarly if \( A \) appears elsewhere followed by a nonterminal \( C \), a state containing \([ [p_1; c_1], [p_1; c_2], \ldots, [p_1; c_n] \] arises (where \( A \Rightarrow c_1 \beta \)). Each of these states can spawn others which include triples \([ p, j; b_i ] \) and \([ p, j; c_k ] \) respectively with \( j>1 \).

Thus a nonterminal which appears many times in the RHS of the productions of a grammar generates many states to keep track of the many different right contexts of its use. Only when \( j=n_p \) (i.e., all characters on the RHS of production \( p \) have been recognised) are the third components of the triples in a state used. If in a state containing \([ p, n_p; b_i ] \), the next character on the input string is \( b_i \); this is taken to imply that production \( p \) did indeed generate the preceding portion of the input string. If \( j \neq n_p \), then in the two states \([ [p, j; b_1], \ldots, [p, j; b_r] \] and \([ [p, j; c_1], \ldots, [p, j; c_m] \] the same set of input symbols will be permissible; each input symbol in this pair of states will cause transition to the appropriate one of a pair of successor states. Each such pair of states could be replaced by a single state were it not for the possibility that the terminal symbols which follow \( A \) in one context (and thus signal the use of some production \( p \) in a parse) may overlap with the set of terminal symbols which signal that \( p \) has not been used in some other context.

Consider, for example, the grammar with productions,

1) \( S \to aAc \)
2) \( S \to bBc \)
3) \( S \to bAd \)
4) \( A \to e \)
5) \( B \to e \)
6) \( S \to aBd \)
In the input string 'ace', it is 'c' which signals that 'e' must be parsed as an 'A' using production 4. For the string 'bec', it is 'c' which signals that 'e' must be parsed as a 'B' using production 5 and not as an 'A' using production 4.

The preceding discussion motivated the following algorithm which utilises the left contextual information inherent to the \( p,j \) components of the \([p,j;\alpha]\) elements of the LR(k) states. Thereafter, like the very successful precedence methods, it relies upon there being no intersection between the set of terminal symbols which imply the use of a certain production in one context with the set of terminal symbols precluding its use in another context. The class of grammars to which the algorithm applies (and which it implicitly defines) are called SLR(k) (Simple LR(k)) grammars. These grammars were defined independently by DeRemer (1969); for consistency his nomenclature is used. The present formulation offers a different viewpoint and includes algorithms for constructing efficient SLR(1) parsers. The basic algorithm is described only for the practical case \( k=1 \), which is somewhat simpler than the algorithm for general \( k \), both conceptually and computationally.

3.1 The SLR(1) Algorithm

Let \( G=(V_u,V_T,P,S_L) \) be a context free grammar. The \( p^{th} \) production is denoted by

\[
A_p \rightarrow X_{p1},X_{p2}, \ldots, X_{pn}, \quad 1 \leq p \leq s \quad \text{and} \quad n_p \geq 0
\]  

(1)

The sets

\[
T_p = \text{follow } (A_p) \cap V_T
\]

(2)

are computed as a preliminary step before commencing the algorithm. (See appendix).

A state \( \lambda_i \) is a set of elements \([p,j]\), where the occurrence of an element in a state means informally that potentially \( j \) characters on the RHS of production \( p \) have been recognised.

The start string \( S_L \) is treated as though it were the RHS of a zeroth production in the grammar, so the initial state \( \lambda_0 = \{[0,0]\} \) implies that no character of the start string has been recognised. The algorithm involves maintaining a stack throughout the parse which is represented (to the left of the vertical bar) as

\[
\lambda, X, \lambda, X, \lambda, \ldots, X, \lambda
\]

(3)

8.
$yeV_r$ denotes the next input symbol and $w\epsilon V_r^*$ represents the (unexamined) remainder of the input string. Initially, the stack contains the single element $A_\epsilon$. Assume that a parse has reached the state $A_m$ so that the stack and input are represented by (3).

The steps involved in proceeding to the next state $A_{m+1}$ are as follows.

**Step 1.** Compute $A_m '$ from $A_m$ where $A_m '$ is defined recursively as the smallest set satisfying

$$A_m ' = A_m \cup \{ [q,o] \mid \exists [p,j] \in A_m, j < n_p, X_p(j+1) = A_q \}$$

i.e. if $[p,j]$ is in the set, the $j+1^{st}$ character, $X_p(j+1)$, of production $p$ is potentially about to be recognised; if $X_p(j+1)$ is a nonterminal symbol defined by, for example, production number $q$, then the first character of production number $q$ could be about to be recognised, so $[q,o]$ is added to the set to reflect this.

**Step 2.** Compute the following sets of terminal symbols

$$Z = \{ aeV_r \mid \exists [p,j] \in A_m, j < n_p, a = X_p(j+1) \}$$

$$Z_p = T_p \text{ if } [p,n_p] \in A_m$$

$$= \emptyset \text{ otherwise.}$$

The set of terminals $Z$ is the set which might legitimately be encountered on the input string for which the sequence of characters at the top of the stack do not correspond to the RHS of any production. $Z_p$ is the set of terminals which signal that the sequence of characters, $X_m - n_p - 1$, $X_m$, on top of the stack matches the RHS of the $p^{th}$ production.

Provided that

$$Z_p \cap Z_q = \emptyset \text{ for all } p,q,$$

an unambiguous choice of action can be made depending upon the next character on the input string.

1) If $yeZ$, stack $y$ so that the stack-input representation becomes

$$A_\epsilon X_1 A_1 \ldots X_m A_m y | w$$

and rename the stack-input symbols producing

$$A_\epsilon X_1 A_1 \ldots X_m A_m X_{m+1} | y w'$$

i.e. $y$ is renamed $X_{m+1}$ and the first symbol of $w$ becomes the new $y$.  

9.
2) If \( y \in \mathcal{Z}_p \), let \( r = m - n_p \). On removing the top \( n_p \) \((X_i \mathcal{A}_i')\) element pairs from the stack and inserting \( \Lambda_p \), the stack-input representation becomes

\[
\mathcal{S}_m X_i \mathcal{A}_i \ldots X_r \mathcal{A}_r \Lambda_p | yw
\]

which after renaming symbols is

\[
\mathcal{S}_m X_i \mathcal{A}_i \ldots X_r \mathcal{A}_r \Lambda_p | yw
\]

(7b)

Step 3. Using the \( \mathcal{S}_m \) defined by (7a) or (7b) compute \( \mathcal{S}_m' \) as in step 1 (or recall it) and insert \( \mathcal{S}_m' \) on the stack where

\[
\mathcal{S}_m' = \left\{ [p, j+1] | [p, j] \in \mathcal{S}_m', j < n_p, X_{m+1} = X_{p(i+1)} \right\}
\]

(8)

i.e. the symbol \( X_{m+1} \) left on the stack in step 2, has just been recognised. If \([p, j] \in \mathcal{S}_m' \) and the \((j+1)\)th symbol on the RHS of production \( p \) matches \( X_{m+1} \), then \([p, j+1] \) is included in \( \mathcal{S}_m' \) to reflect the fact that, in the next state, potentially \( j+1 \) symbols on the RHS of production \( p \) have been recognised.

If \( \mathcal{S}_m' = \{[0, 1]\} \), the parse is complete; otherwise an inductive step on \( m \) has been completed, and the algorithm proceeds from step 1. This method of terminating the algorithm requires that the state whose set denotation includes the member \([0, 1]\) has only this member; this implies the very weak constraint that the start symbol of the grammar must not be left recursive.

The description of the algorithm deliberately parallels that of Knuth's LR(k) algorithm in order to facilitate comparison. We observe that

1) The set (5a) corresponds to the set (19) in Knuth's LR(k) algorithm (taking \( k = 1 \) in the latter) but it is appreciably less costly to compute. In essence, (5a) differs in simply ignoring the cases where \( X_{p(i+1)} \) are nonterminal symbols; this is permissible since the closure computation (of \( \mathcal{S}_m \) to \( \mathcal{S}_m' \)) in step 1 ensures that if \([p, j] \in \mathcal{S}_m \), then \( \mathcal{S}_m' \) contains all possible \([p', j']\) such that

\[
X_{p(i+1)} = X_{p'(j'+1)} \omega, \text{ where } \omega \in \mathcal{V}^*_k.
\]

It follows that, in determining the single symbol terminal strings derived from the symbol \( X_{p(i+1)} \), attention can be restricted to those symbols \( X_{p'(j'+1)} \) which are terminals. This same observation, for general \( k \) values implies that the set (19) in the LR(k) algorithm can be replaced by

\[
Z = \{ [p, j] | \mathcal{S}_m' \}, \mathcal{S}_m', j < n_p, \mathcal{S} = \mathcal{A}_p, \mathcal{A} = X_{p(i+1)} \mathcal{A}_{p(i+1)} \}
\]

when \( k \) is other than one, the computational simplification is not so great.

2) The only difference between the SLR(1) algorithm and the LR(1) algorithm lies in replacing set (20) of the latter by (5b) above.
3) A grammar (and hence the language it defines) is SLR(1) if the conditions (6) hold for all states $\lambda$, which can arise in parsing sentences of the language.

4) The SLR algorithm retains two important properties of the LR algorithm. Syntactic errors are located on the first terminal of a sentence which is not generated in accordance with the rules of the grammar. Such errors become apparent in step 2 of the algorithm in that $\gamma \neq \Sigma$ nor does $\gamma \neq \Sigma'$, for any $p$. The second feature is that when the stack is reduced in step 2, it is known, a priori, that the characters on the top of the stack do indeed correspond to those of the RHS of the production. Their presence on the stack is redundant; in a practical algorithm they are omitted. Relation (8) shows that $X_{m+1}$ (which is in consequence termed the associated symbol of the state) is in any event, uniquely associated with $\lambda_{m+1}$.

Korenjak (1969) has described a method, closely related to the SLR method, for reducing the number of states generated by the LR algorithm. In it, the LR algorithm is used to construct sub-parsers for certain selected nonterminals of a grammar. The main parser for the grammar is also constructed by the LR algorithm, but special provision is made for invoking the sub-parsers whenever the corresponding nonterminals are to be recognised. In that each sub-parser must work for all right contexts of its nonterminal in the RHS of the productions of the grammar, it is not surprising that the SLR algorithm is equivalent to Korenjak's if, in the latter, sub-parsers are constructed for all nonterminals of the grammar.

3.2 SLR(1) State Tables and the Basic Parsing Algorithm

The SLR(1) algorithm has been described in terms of parsing a given input string. It can be modified to produce state tables which, with a simple interpreter and a stack, suffice to parse all strings derived according to the grammar. To produce the state tables, a list of all possible states is maintained and $m$ becomes the index to states in this list. (Initially the list contains only $\lambda = \{[0,0]\}$ and $m = 0$). Step 1 of the algorithm is unchanged; Steps 2 and 3 are replaced by the following.

Step 2. Compute

\[
Z' = \{ X \in V \mid A_\lambda[j, j] \in \Sigma', j < n_p, X = X_p \sigma_{i+1} \} \]

(58)

$Z'_p$ as in (5b)
(Note that the members of $Z'$ in (5a') range over the entire vocabulary. The members of $Z$ in (5a) were restricted to $V_T$. In fact 

$$Z = Z' \cap V_T$$

and conditions (6) must still hold.)

**Step 3.** For each symbol in $V$ a state table entry for the state $S_m$ is determined from:

- If $X \notin Z'$, for any $p$, the entry is blank,
- if $X \in Z'$, the entry is the production number $p$,
- if $X \in Z'$, the entry is the state $S_x$ to be entered next where 

$$S_x = \left[ [p, j+1] [[p, j]] \right]_{S_m, j<n_p, \text{X=X}_{P_{m+1}}} \{ \right.$$

If $S_x$ is not present in the list of all possible states it is added. If $m$ is not pointing to the last state in this list, it is incremented by one and a return is made to step 1.

Blank entries in the state table can only be encountered when parsing syntactically invalid strings. A production number entry when encountered implies popping a number of symbols from the stack corresponding to the number of symbols on the RHS of the production then scanning the LHS symbol. The purpose of next state entries is obvious.

An example of a state table produced by this modified form of the SLR(1) algorithm is given in figure 2; the productions of the grammar to which it corresponds are shown in figure 1. In figure 2, production numbers are prefixed by a minus sign. For indicating next states, the denotation as a set of $[p, j]$ elements is inconvenient, the index number of the state in the list produced by the modified algorithm replaces it.

The following procedure, which follows immediately from the construction of the state tables, suffices to interpret them. Assume that state $i$ is the current state and symbol $j$ is being scanned (initially $i=0$, corresponding to $S_0 = \left[ [0, 0] \right]$) and $j$ is the first symbol on the input string); then

**Step 1.** If $j$ is a terminal symbol then stack $i$ on the parse stack.

**Step 2.** Examine the $(i, j)^{th}$ entry of the state table, if the entry is

- a) blank, then symbol $j$ is an invalid symbol and an error handling routine is invoked,
- or b) a production number, then one stack element is popped from the parse stack for each character on the RHS of this production thereby uncovering a state number, $k$; $j$ is stored in a temporary
location TEMP and the LHS of the production is a nonterminal, \( I \).

or c) a state number, then this state number, \( k \), is recorded. If \( j \) is a terminal symbol, then the next input symbol, \( l \), is read otherwise \( l \) is assigned from TEMP.

**Step 3.** \( k \) and \( l \) replace \( i \) and \( j \) respectively and a return is made to step 1.

A convenient termination of this procedure can be achieved by detecting the next state number entry corresponding to the state \([0,1] \) in step 2c.

The rather small example of figure 1 conveys no impression of the size of the tables which would result for a practical grammar; examples will be given in section 6.

### 3.3 The LALR(1) Algorithm

Consider the grammar with productions

1. \( S \rightarrow \alpha A \delta \)
2. \( S \rightarrow \alpha cc \)
3. \( S \rightarrow \beta \delta c \)
4. \( A \rightarrow c \)

On attempting to produce an SLR(1) state table for this grammar as described in 3.2, a state \([2,2],[4,1] \) will be generated. For this state, \( Z = \{c\} \) and \( Z_4 = \{c,d\} \), hence \( Z \cap Z_4 \neq \emptyset \), demonstrating that the grammar is not SLR(1). However, if in state \([2,2],[4,1] \) with input symbol \( y = c \) the stack is reduced using production 4, it must then contain \( \alpha A \). Since \( \alpha A c \) cannot begin any sentential form in the language, we could (and should) take \( Z_4 = \{d\} \) in this instance.

More generally, consider a state \& for which \( \alpha \in T_p \) with input symbol \( y = \alpha \). Denote by \( \alpha \) the character string on the stack after reduction by production \( p \). If for all such \( \alpha \), there is no \( \omega \in T_p \) such that \( S \gamma \omega = \omega \) then we should delete \( \alpha \) from \( Z_p \). (Note that no symbols can be deleted from \( Z \) since this would imply the possibility of reading past an error in the input string.)

This inaccuracy in the sets \( Z_p \) of the SLR(1) algorithm is due to the use of the sets \( T_p \) to determine the symbols which indicate reduction of the stack, instead of that subset of \( T_p \) which is the permitted right context of production \( p \) in a particular state.

\( T_p \) is the permitted right context of production \( p \) over all states.

In the above example, the state \( [[2,2],[4,1]] \) can only be reached in parsing a string generated by production 1 or 2;
production 3 cannot be involved so that in the context of this state only the subset \( \{d\} \) of \( T_4 = \{c,d\} \) can follow the use of production 4.

The example also demonstrates that a larger class of grammars than SLR(1) can be parsed using a state table in which \( Z_p \) is computed correctly rather than substituting \( T_p \). This larger class is termed LALR(1) following DeRener. It should also be pointed out that LALR(1) state tables may be more economical of storage under some representations e.g. the list representation discussed in 5.1.

The one disadvantage of LALR(1) compared with SLR(1) is that computation of the state tables of the former is much more complex though clearly less effort is involved than in computing LR(1) tables.

Two alternative methods for constructing LALR(1) tables will be briefly presented here.

LALR(1) state tables may be computed directly. To do this the LR(1) notion of a state as a set of triples \([p,j;a]\) as discussed in 3 is needed. So also are the original Knuth versions of equations (4) and (5b) for \( k = 1 \) i.e.

\[
\delta_m' = \delta_m \cup \{ [q,o;b] | \exists [p,j;a] \in \delta_m', j < n_p, X_p(j+1) = A_q' \}
\]

\[
Z_p = \{a | [p,n_p;a] \in \delta_m' \}
\]

If formulated to parse a given input string there would be no difference between an LALR(1) parser and an LR(1) parser. The difference arises in constructing the state table. The determination of whether a state is already in the list of possible states is made only on the basis of the \( p \) and \( j \) components of the triples, i.e. as in an LR(0) computation. Two sets of triples belong to the same state and are merged if for each triple \([p,j;a]\) in one set there is a triple \([p',j';a']\) in the other set such that \( p = p' \) and \( j = j' \). Such merging may introduce new triples (differing only in the third component); if so then the augmented state is marked. When the list of states is completed, each marked state is treated in turn, by first removing the mark, then recomputing the set of possible next states. (The mark on a state indicates that its right context has been enlarged with the consequence that the right context of some of its next states become enlarged). The recomputed next states are merged with their original versions marking any which become augmented. The reprocessing of marked states is continued until all marks have been removed.
Figure 1

\[
A \rightarrow id := E \\
B \rightarrow if B then A L \\
T \rightarrow E + T \\
L \rightarrow \text{else } S \\
L \rightarrow A
\]

Figure 2

\[
\begin{array}{cccccccccccc}
\text{id} + ( ) & * & \text{if} & \text{then} & \text{or} & \text{else} & := & \bot \\
& & 1 & 2 & 3 & & & & \\
4 & 5 & & & & & & & & \\
0 [0,0] & & & & & & & & & & \\
1 [0,1] & & & & & & & & & & \\
2 [1,1] & & & & & & & & & & \\
3 [2,1] & & & & & & & & & & \\
5 [4,1] & & & & & & & & & & \\
6 [3,2] & & & & & & & & & & \\
7 [4,2], [11,1] & & & & & & & & & & \\
8 [12,1] & & & & & & & & & & \\
9 [3,3], [6,1] & & & & & & & & & & \\
10 [5,1], [8,1] & & & & & & & & & & \\
11 [7,1] & & & & & & & & & & \\
12 [9,1] & & & & & & & & & & \\
13 [10,1] & & & & & & & & & & \\
16 [6,2] & & & & & & & & & & \\
17 [8,2] & & & & & & & & & & \\
18 [6,1], [9,2] & & & & & & & & & & \\
21 [6,3], [8,1] & & & & & & & & & & \\
23 [9,3] & & & & & & & & & & \\
26 [13,2] & & & & & & & & & & \\
\end{array}
\]
(Termination is ensured since the right context of each state must increase to be marked, and $V_\text{f}$ is finite.) This method implicitly uses LR(0) states to form equivalence classes of the states in an LR(1) state table, but the computation is not as large as full LR(1), since not every LR(1) state is necessarily generated.

A second method is to first form an SLR(1) table and then eliminate the invalid members of the $Z_\text{p}$ sets as follows. Define an $n^{th}$ predecessor of a state $\Delta$, as any state from which an $n-1^{th}$ predecessor of $\Delta$ can be obtained as a possible next state, and take $\Delta$ to be its own 0$^{th}$ predecessor. We denote the exact right context of $[p,j]$ in a state $\Delta_i$ by $R(\Delta_i,[p,j])$, which is defined recursively by

$$R(\Delta_i,[p,j]) = \{ \text{as first } (X_{q_1}X_{q_2} \cdots X_{q_k} b), [q,k] \in \mathcal{L}, X_{q_k} = \lambda_\text{p}, \text{be } R(\Delta_{q_i},[q,k]), \Delta_{q_i} \text{ is a } j^{th} \text{ predecessor of } \Delta_i \}$$

This definition provides a finite algorithm if circularity is avoided by taking

$$R(\Delta_i,[r,1]) = \emptyset$$

whenever a computation of $R(\Delta_i,[r,1])$ invokes itself. The exact right context of $[p,n_r]$ in a state may now be used as $Z_\text{p}$ for that state, giving an LALR(1) state table.

It may be more practical only to attempt to refine $Z_\text{p}$ in the cases where $Z_\cap Z_\text{p} \neq \emptyset$ or $Z_\text{p} \cap Z_\neq \emptyset$. This would achieve acceptance of all LALR(1) grammars without necessarily performing a full LALR(1) computation. Lalonde (1971) has used this technique to refine every state which is not LR(0).

4. Transforming the State Tables

A number of transformations can be applied to state tables produced by algorithms of the LR type with a view to reducing their space requirements or improving the performance of the parsing process based upon them. Conceptually, at least, these transformations may be envisaged as being applied to the matrix representation of the state tables; modification of the constructor algorithm may be more convenient.

For practical purposes, reduction of the number of states (rows of the matrix) by general state minimisation techniques (Pager, 1970) can be discounted immediately; the combinatorial nature of these techniques militates against their use; even SLR(1) tables typically involve a few hundred states for programming language grammars. Recourse to these techniques
also neutralises some of the benefits of the LR methods with respect to error detection. It is necessary to develop special techniques taking into account features of the LR algorithms and general characteristics of programming language grammars to reduce the size of the state tables substantially.

It is appropriate to recall that, in LR type state tables, the blank entries in columns corresponding to nonterminal symbols are not significant; all syntactic errors are detected by encountering a blank entry in a column associated with a (terminal) input symbol. We observe then that merging states with nonconflicting nonterminal entries is possible but that merging states with other than identical entries for terminal symbols affects the error detection capability. Merging states with nonconflicting terminal entries is possible with delayed error detection if the parsing algorithm is modified to include checking that the associated symbols of the states on the top of the stack match the RHS of the appropriate production every time a production number entry is encountered. Detection of errors on the first input character to depart from the rules of syntax is a useful feature of LR methods. Apart from possible benefits in the error recovery problem, compilers using delayed error detection are prone to failures in semantic routines in the interval between reading the invalid character and the error being detected by the syntax analyser. Attention has been intentionally confined to transformations preserving this immediate indication of errors.

4.1 Elimination of LR(0) Reduce States and Chain Derivations

In figure 2, the rows 1,2,3,8,11,13,20,22,23,24 and 26 have the LR(0) property that it is not necessary to inspect the input symbol to determine the necessary action since there is a unique action in each of these states, namely, reducing the stack using a production number which is associated with all valid entries in these states. Failure to inspect the input symbol on any production number entry in the table causes no significant delay in error detection; the subsequent action of the parser will be to use the LHS of the production to determine the next action of the parser. Ultimately a next state entry is encountered when this input symbol must be inspected. It follows that states containing a unique reduce action can be eliminated by the simple device of replacing the next state entries in the table causing transition into such states by the appropriate production
Some of the states which would be eliminated by this change, notably those corresponding to rows 1, 2, 3 and 11 in figure 2 are involved in a more general transformation which is suggested by considering the problem of eliminating chain derivations. The need to group syntactic constructs together to form a further syntactic construct results in frequent occurrences of productions of the form $A \rightarrow B$, where $A, B \in V_N$. The specification of precedence of operations in arithmetic expressions, for example, also introduces productions of this type, and derivations of the form

$$\alpha A_1 \beta = \alpha A_2 \beta = \ldots = \alpha A_n \beta \Rightarrow \alpha \gamma \beta,$$

where $A_1, A_2, \ldots, A_n \in V_N$ result very frequently. Usually, in compilers, only the final step involving the production $A_n \rightarrow \gamma$ has semantic significance and a parser which bypasses the other steps would be advantageous.

The example grammar admits the chain $E \Rightarrow T \Rightarrow P$ and there are two states (rows 6 and 12 of figure 2) in which an instance of $E$ is to be recognised. Considering row 6, it is observed that when a $P$ has been recognised, a transfer is made to row 11 where production 7 causes $P$ to be parsed as $T$, then on return to row 6, the $T$ entry implies a transfer to row 10 where, if the input symbol is $'x'$, a transfer is made to row 17 otherwise production number 5 must be applied causing $T$ to be parsed as $E$. Row 6 then indicates that for $E$ a transfer is made to row 9. At this stage it would emerge that $'1'$ is an invalid successor to the present instance of an $E$; it is valid however in row 12. We observe that, if there is no semantic significance attached to production 7, then row 11 may as well not exist, a transfer to row 10 would suffice; in row 10, if there is no semantic significance attached to production 5, then the $-5$ entries simply serve to delay an error signal until row 9 is entered. The redundant references to the $-5$ entries could be eliminated by replacing $-5$ entries in row 10 by the entries in the corresponding columns of row 9 (omitting the $-5$ entry in the '1' column) were it not for the fact that row 10 is entered from row 12 also. By a similar argument the entries from row 18 should replace $-5$ entries in row 10 (omitting those in the 'else' and 'if' columns). These assignments are incompatible. If no entries are omitted from row 10, the assignment incompatibility disappears but so also does
the error detection capability. Complete elimination of chain derivations is possibly only at the price of delayed error detection (matching stack symbols with a production RHS whenever the stack is reduced) or at the expense of introducing extra states, e.g., if in our example state 10 is split into two copies row 9 can use one and row 12 the other.

It is clear that states containing a single \([p,j]\) element can be eliminated if \(j=n_p\). A special case arises whenever \(n_p=1\), \(X_p \in \mathcal{V}_m\) and production \(p\) has no semantic significance. In this case, all reference to production \(p\) may be deleted. (The conditions \(n_p=1\), \(X_p \in \mathcal{V}_m\) imply that production \(p\) is a chain production; future usage of this term will imply the additional requirement that no semantics are associated with this production). The changes to the state tables are most readily described in terms of further modifications to the constructor algorithm. In the modified SLR(1) algorithm, step 2 computes a set \(Z'\) and sets \(Z_p\) for each state. For each member of a set \(Z_p\), the state table contains a production number entry \(-p\) in the corresponding column. For each member of \(Z'\), step 3 computes the appropriate next state entry and adds any new states to the list of states. Only step 3 is modified to accommodate the required changes.

**Step 3** Only the treatment of \(Z'\) changes.

Let \(Z' = \{X_1, X_2, \ldots, X_r\}\)

Compute the sets \(W_t = \{[[p,j+1], 3[p,j] \in \mathcal{A}_m, j<n_p, X_c = X_p(j+1)\}, 1 \leq i \leq r\}.

The sets \(W_t\) are the new states of the SLR(1) algorithm. The elimination of all intermediate steps in a chain derivation can be achieved by computing, for \(t=1,2,\ldots,r\), the smallest set

\[
W_t' = W_t \cup \{\} \quad \text{if} \quad \text{chain production, } A_p = X_c
\]

then, for each \(t\), compute

\[
W_t'' = W_t' - \{[[p,1]], [[p,1]\in V_t' \quad \text{if} \quad \text{chain production}\}.
\]

Then members of \(W_t'' \backslash W_t'' \neq \{[[p,n_p]]\}\) not already present are added to the list of states.

Any \(W_t''\) consisting of a single \([p,n_p]\) element would produce an LR(0) state in which the only valid action would be to reduce the stack using production \(p\). Such a set \(W_t''\) is precluded from generating a new state and instead the entry in the state table for state \(\mathcal{A}_m\) and symbol \(X_c\) is made to specify a reduction of the stack using production \(p\). If \(X_c \in \mathcal{V}_t\), then \(X_c\) must (conceptually) be added to the stack before the reduction occurs. (In practice a
<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4,2</td>
<td></td>
<td>[11,1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3,3</td>
<td></td>
<td>[6,1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8,1</td>
<td></td>
<td>[3,3], [6,1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8,1</td>
<td></td>
<td>[6,1],[9,2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>9,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>11,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>6,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>8,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>6,1</td>
<td></td>
<td>[9,2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>4,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>6,3</td>
<td></td>
<td>[8,1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>13,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{id} + ( ) * \begin{cases} \text{if then or else} \end{cases} := \downarrow S \ A \ I \ E \ T \ F \ B \ L
\]

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*12</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>9</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>16</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>16</td>
<td>17</td>
<td></td>
<td>-3</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>16</td>
<td>*9</td>
<td>17</td>
<td></td>
<td>-3</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>18</td>
<td>10</td>
<td>10'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*11</td>
<td></td>
<td></td>
<td></td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>21</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>16</td>
<td>*9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>25</td>
<td>-14</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10</td>
<td>12</td>
<td></td>
<td></td>
<td>-6</td>
<td>-6</td>
<td>17</td>
<td></td>
<td>-6</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td>-13</td>
<td>-13</td>
<td>-13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3
Further symbol must be read from the input string \( i.e. \) if \( X_e \in V_n \), an ordinary production number entry, denoted by \(-p\) in figure 2, is employed; if \( X_e \in V_r \) a new type of entry is used, a **scan-production number** entry which will be denoted \(*p*\).

The computations of step 3 are illustrated in the case of \( \mathcal{A}_m = \{[3, 2]\} \) for the example grammar of figure 1. (See also rows 6, 9, 10, 11, 12 and 13 of figure 2). The set \( \mathcal{Z}' = \{id, (, E, T, P)\} \), thus

<table>
<thead>
<tr>
<th>( t )</th>
<th>( X_e )</th>
<th>( W_e )</th>
<th>( W_e' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([10, 1])</td>
<td>([10, 1])</td>
<td>([10, 1])</td>
</tr>
<tr>
<td>2</td>
<td>([9, 1])</td>
<td>([9, 1])</td>
<td>([9, 1])</td>
</tr>
<tr>
<td>3</td>
<td>([3, 3],[6, 1])</td>
<td>([3, 3],[6, 1])</td>
<td>([3, 3],[6, 1])</td>
</tr>
<tr>
<td>4</td>
<td>([5, 1],[8, 1])</td>
<td>([5, 1],[8, 1],[3, 3],[6, 1])</td>
<td>([8, 1],[3, 3],[6, 1])</td>
</tr>
<tr>
<td>5</td>
<td>([7, 1])</td>
<td>([7, 1],[5, 1],[8, 1],[3, 3],[6, 1])</td>
<td>([8, 1],[3, 3],[6, 1])</td>
</tr>
</tbody>
</table>

The appearance of \([5, 1]\) in \( W_e \) implies (since production 5 is the chain production \( E = T \)) that \( W_{e1} \mid W_{e3} \) constitutes \( W_e' \). The appearance of \([7, 1]\) in \( W_e \) implies (since production 7 is \( T = P \), a chain production) that \( W_e \) must be included in \( W_e' \) with \( W_e \). This introduces \([5, 1]\) into \( W_e' \), which implies that \( W_{e3} \) must also be included. The \( W_j' \), \( 1 \leq j \leq 5 \), result from \( W_j \) on removing \([5, 1]\) and \([7, 1]\) from \( W_e' \) and \( W_e' \).

Figure 3 shows the state table for the example grammar resulting from the above modifications to the constructor algorithm.

Some steps in chain derivations may be eliminated without increasing the number of states by taking

\[
W_e' = W_e
\]

where \( X_e = X_u = X_v = \ldots = X_e \) is the longest chain derivation for which \( W_{e1}, W_{e3}, \ldots W_{en} \) and \( W_e \) each have a single member. In terms of the previous discussion relating to figure 2 this modification eliminates states like \([7, 1]\) in row 11 while avoiding the replication of \([5, 1],[8, 1]\) in row 10, which is entailed in eliminating the chain production 5.

A special case which is not covered by the modifications is the case of productions with a null RHS. Usually semantic information will be associated but if not, for such a production \( p \), the state table entries for the members of \( Z_p \) are replaced by the entry for \( A_p \) in the same row of the table. In figure 3 the effect of this would be to copy the \(-4\) entry in the \( L \) column of row 19 into the \( L \) column to replace the existing \(-14\) entry.
To accommodate the addition of a new type of entry in the state table, the parsing algorithm must be amended by the inclusion, following step 2c) of:

- a) a scan production entry, then one fewer than the number of symbols on the RHS of production \( p \) are popped from the stack uncovering a state number \( \kappa \); the LHS of the production is a nonterminal symbol assigned to \( l \); the next input symbol is read and stored in TEMP.

4.2 Connection with Operator Precedence Methods

The connection with operator precedence methods is considered here for its intrinsic interest. It suggests a method of reducing the number of rows and possibly columns of the state tables of the LR methods. No attempt has been made to utilise the connection practically, but it could prove useful in the absence of hardware facilities enabling efficient use of the list representations to be described.

For expository purposes and because it is implicit in the operator precedence methods, we temporarily relax the constraint that errors be detected on the first invalid symbol from the input string. On the basis of the preceding discussion (section 4.1), this would, for example, permit rows 9, 10, 10' and 18 of figure 3 to be merged. Also, it is assumed that row 19 has been modified to remove reference to production 14 as described, so that the state table has the form shown in figure 4.

The example grammar of figure 1 is not an operator grammar but it satisfies the more general condition (see Gray and Harrison, 1969);

\[
    S \rightarrow aABV, ABcV, \text{ and } a, cV, \text{ then first } (B) \subseteq V_T. \tag{10}
\]

If the associated symbol of a state \( \delta_j \) is a nonterminal symbol, then condition (10) implies that row \( j \) of the state table has entries only in the terminal symbol columns. Thus, if row \( i \) contains a next state entry designating row \( j \), then merging rows \( i \) and \( j \) of the table can cause no conflict of entries in the nonterminal columns of the two rows. If \( A \) is the associated symbol of \( \delta_j \), then the requirement that

\[
    \text{first } (A) \cap \text{follow } (A) \cap V_T = \emptyset \tag{11}
\]

ensures that no conflicts can occur in terminal symbol columns either.
Figure 4

<table>
<thead>
<tr>
<th>id + ( ) *</th>
<th>if then or else := =</th>
<th>S</th>
<th>A</th>
<th>I</th>
<th>E</th>
<th>T</th>
<th>P</th>
<th>B</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*12</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td></td>
<td>14</td>
<td>15</td>
<td>9 9 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*9 17</td>
<td>-3</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td></td>
<td>9 9 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td>21 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td>-8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 17</td>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5

<table>
<thead>
<tr>
<th>id + ( ) *</th>
<th>if then or else := =</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>-0</td>
</tr>
<tr>
<td>*12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>*10 12</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*10 12</td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-13 -13 -13</td>
<td></td>
</tr>
</tbody>
</table>
It is precisely the fact that operands satisfy this condition which precludes their confusion with operators in Gray and Harrison's extension of precedence methods; also the SLR(1) algorithm implies that row $i$ contains entries only in columns corresponding to \text{first}(A), while row $j$ entries are confined to columns corresponding to \text{follow}(A).

Consequently, if a grammar satisfies conditions (10) and (11) and if row $i$ of its SLR(1) state table contains a next state entry in a nonterminal column referring to row $j$, then row $j$ can be eliminated by merging it with row $i$. If a row $i$ contains a production number entry in a nonterminal column, this entry arose by eliminating a row $j$ in which all entries referred to this production or alternatively it arose by eliminating a chain derivation in which several rows were combined to form an effective row $j$ in which all entries referred to this production. In either case condition (11) permits row $j$ to be merged with row $i$.

Assuming that all nonterminal entries in a row $i$ are considered and the corresponding $j$ rows are merged and that this is repeated for all rows $i$, it is now clear that the nonterminal columns of the state tables are redundant. During a parse, assume that reducing the stack, using production $p_i$, causes a return to row $i$. Instead of inspecting the column corresponding to $A_i$, it is only necessary to inspect the column corresponding to the current input symbol (where the appropriate action from the merged row $j$ will be encountered. Matching the RHS of a production with the top of the stack on each reduction of the stack is of course necessary to preserve error detection.

Figure 5 shows the outcome of these transformations on the state table in figure 4. The associated symbol of $\lambda_i$ for each row $i$ is also shown. The underlined entries are those which have been merged in. The state table in figure 5 corresponds very closely to the transition matrix resulting from Gries' augmented operator precedence method (1968). It differs significantly in two respects. Operator precedence methods do not eliminate LR(0) states containing only production number entries. (Note that figure 5 contains no state for which the associated symbol is '(','). The second difference lies in the fact that Gries' transformation of the original grammar introduces 'starred nonterminals' which correspond only approximately to the 'states' $\lambda_i$ in figure 5. (The approach described here if applied to Gries' example grammar would not introduce the unnecessary state $<\text{expr} - \text{or} - \text{var}>$ in figure 7 of Gries' paper).
Even if a programming language grammar is not an operator grammar, it appears that several nonterminal symbols satisfy
\[ \text{first}(A) \cap \text{follow}(A) = \emptyset \]  
(12)
Furthermore in the SLR(1) algorithm this test is conveniently made. The sets \( T_p \) are most easily obtained from \( T_p = \text{follow}(A_p) \cap \mathcal{V}_T \) and the sets \( \text{follow}(A) \) in turn can be computed from \( \text{first}(A) \) for all nonterminals \( A \) so that these sets are available for testing condition (12). (See appendix).

Any row \( j \) for which the associated symbol is a nonterminal \( A \) satisfying (12) can be merged with a row \( i \) containing a next state reference to row \( j \) in its \( A \) column if this reference is replaced by a special flag entry and each entry from the terminal columns of row \( j \) is tagged with a one bit marker on merging them into row \( i \). The flag and the markers suffice to preserve error detection. Detection of the special flag entry by a parsing algorithm would imply that the current input symbol be used to select an entry from the row of the state table containing the flag entry; the selected entry would be valid if tagged. In all other circumstances only the untagged entries would be valid. Row \( j \) can only be deleted if it can be merged into all rows \( i \) which cause a transfer to it; also if row \( i \) refers to rows \( j \) and \( j' \) in its nonterminal columns it may not be possible to merge both rows \( j \) and \( j' \) into row \( i \).

### 4.3 Simple Row Merging and Partitioning of State Tables

The state table in figure 3 is quite typical in that many of the states have no entries in the columns corresponding to nonterminal symbols; states with this property can be renumbered to occur in adjacent rows at the end of the state table. An obvious space saving results by performing this renumbering then partitioning the state table into two arrays; the \( T \)-table contains only terminal symbol columns and the \( N \)-table contains only the nonterminal symbol columns. Figure 6 results from figure 3 using the state renumbering scheme:

Old Number: 0 4 5 6 7 9 10 10' 12 14 15 16 17 18 19 21 25
New Number: 1 10 8 3 11 12 13 16 4 9 14 5 6 15 7 17 2

If the state table is separated into a terminal part and a nonterminal part, advantage can be taken of the fact that several rows in the terminal array are identical; at the expense of an array with one element for each state, indirect addressing permits the identical rows to be overlaid. The nonterminal array can be
Figure 6

<table>
<thead>
<tr>
<th>T-Table</th>
<th>N-Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 7.
similarly treated; in this case rows with nonconflicting entries can be overlaid to produce state tables of the form shown in figure 7.

As a final comment on the matrix representation of the state table, it is worth noting that the states can be renumbered so that the next state numbers in a column of the T-table are represented by a sequence of consecutive integers \( a_i \), which can be decomposed into the form \( c = b_i \) where

\[
c = \min (a_i - 1)
\]

The existence of such a renumbering of states follows from the fact that all next state entries causing transition to a particular state lie in one column of the state table, namely, the column corresponding to the associated symbol of the state.

The values \( b_i \) can be stored in the table; \( c \), in an array element corresponding to the column. The requirement that the \( a_i \) be consecutive is not compatible with the renumbering to partition the state table; it is sufficient, however, if the range of \( b_i \) values in any column is small compared with the number of states in the state table; then fewer bits are needed to encode the \( b_i \) than the state numbers. Advantage can only be taken of this if production number entries can also be compactly encoded. It is not possible, in general, to renumber productions so that those in a row of the T-table have consecutive numbers, but it should be possible to keep the range of production numbers in a row small compared with the total number of productions. No attempt has been made to pursue this possibility.

5. List Representation of the State Tables

The motivation for representing state tables by means of lists is twofold: one is the obvious space saving resulting from the elimination of blank entries; the other is to exploit the list searching instructions available on modern computers.

The IBM System/360 "translate and test" instruction, for example, permits the searching of a list (the A list) of 8 bit characters for the first occurrence of a member of a second list (the B list). Its result is, in effect, either the index of this first element or an indication that no element in the A list matches an element of the B list.

5.1 Encoding of the T-table

For row \( i \) of the T-table, two arrays are constructed:
one, TCHAR(i), is a list of the valid input characters for this row; the other, TACTION(i), is the list of corresponding entries from the row of the state table. Given an input character in the B list and taking TCHAR(i) as the A list, the translate and test instruction provides either an index into TACTION(i) for the appropriate state table entry or an indication that the input symbol is invalid.

Production number entries referring to the same production often occur several times in a row of the T-table (see rows 8, 9, 13, of figure 7). Because error detection can be deferred until after the stack has been reduced, it is convenient to omit all entries for one production from TCHAR(i) and TACTION(i) for such a row. Instead a special code, default, representing a symbol not in V is appended at the end of the list TCHAR(i) and the production number is inserted in the corresponding element of TACTION(i). The default code is included in the B list for the translate and test instruction. For grammars more complex than our example, two or more sets of production number entries may exist in a row; one is chosen for the default entry which reduces the number of elements in TCHAR(i) as much as possible.

The arrays TCHAR(i) and TACTION(i) for all rows i can conveniently be stored head to tail in arrays TSYM and TACT. Access to these arrays is achieved via an array TSTATE containing one entry with two fields for each row i. One of these fields indexes the first element in TSYM or TACT for row i the other field specifies the number of entries in these arrays for row i. The compacted form of the T-table shown in figure 7 is a convenient form from which to build the lists. There is now the possibility of further economies in storage e.g. the only entry in row 5 of figure 7 is identical to an entry which must be present in TSYM and TACT for row 1; no additional space is needed for it.

5.2 Compacting the N-Table lists

The general problem of overlapping the entries in TSYM and TACT from rows of the state tables which have common subsets of entries to economise on storage under the restriction that all elements of a row must lie in sequential positions is nontrivial; however, worthwhile storage savings can be achieved with relatively simple strategies. Two which have proved to be quite effective are described.
Let \( A_i = \{a_{i1}, a_{i2}, \ldots, a_{in_i}\} \) denote the elements of TCHAR(i) and \( B_j = \{b_{j1}, b_{j2}, \ldots, b_{jn_j}\} \) denote the corresponding elements of TACT(i). Sufficient (though certainly not necessary) conditions for entries from two rows \( i \) and \( j \) to overlap in the arrays TSYM and TACT are

1. \( A_i \cap A_j \neq \emptyset \),
2. if \( a_{ik} = a_{jk} \in A_i \cap A_j \) then \( b_{ik} = b_{jk} \),
3. \( a_{in_i} \) and \( a_{jn_j} \) are not both equal to the default code.

Since \( A_i = (A_i - A_j) \cup (A_i \cap A_j) \) and \( A_j = (A_j - A_i) \cup (A_i \cap A_j) \), the elements of \( A_i - A_j \), then \( A_i \cap A_j \), and finally \( A_j - A_i \) can be inserted in TSYM in sequence. (As each element \( a_{ik} \) or \( a_{jk} \) is inserted into TSYM, its corresponding element \( b_{ik} \) or \( b_{jk} \) is inserted in the corresponding position of TACT). Condition 3 above ensures that, after perhaps interchanging the roles of the names \( i \) and \( j \), any default code is located in the set \( A_i - A_j \) and this must be the last element of this set to be inserted into TSYM.

No attempt is made to overlap entries from more than two states, and the pairs of states are selected with the aid of a triangular matrix \( C \) for which the elements are defined by

\[
C_{ij} = \begin{cases} 0, & \text{for any pair of rows } i \text{ and } j \text{ not satisfying the three conditions (13),} \\ \text{cardinality of } A_i \cap A_j, & \text{otherwise.} \end{cases}
\]

Pairs of states \( r \) and \( s \) whose entries are to be overlapped are selected by repeatedly determining

\[
C_{rs} = \max_{ij} (C_{ij})
\]

then effectively replacing all entries in rows and columns \( r \) and \( s \) by zeros until no nonzero element remains in the matrix \( C \).

If two states both contain default entries, the conditions for overlapping their entries include the requirement that \( b_{in_i} = b_{jn_j} \); this seems to be rarely satisfied in practice unless chain derivations are eliminated when the technique, described next, is far more effective.

The treatment of chain derivations was illustrated in section 4.1 by considering the derivation \( E = T = P \) in the example grammar of figure 1. The sets \( W_\delta, W_\pi \) and \( W_\omega \) were seen to correspond to the states entered after recognising \( E, T \) and \( P \) respectively. The method of constructing these sets ensures that \( W_\delta \subseteq W_\pi \subseteq W_\omega \), from which it follows that the (nonblank) state table entries for the state \( W_\delta \) are a subset of those for the state
\( W^* \) which in turn are a subset of those for \( W^*_5 \), i.e., in the list representation the entries for the states \( W^*_3 \) and \( W^*_4 \) can both be overlaid on those of \( W^*_6 \). If during the construction of state tables, the state numbers of sets of states like \( W^*_3, W^*_4 \) and \( W^*_6 \) are recorded, they can be used to overlay entries in the arrays TSTM and TACT.

The procedure below is similar in intent but caters for 'accidental' inclusions of the entries of one state within another. The process involves appending a mark to each state; initially all states are unmarked.

**Step 1.** An unmarked state \( \xi_m \) with the maximum number of nonblank entries is selected; then the largest set \( Q_1 \) is computed such that
\[
Q_1 = \{ \xi_j \mid \xi_j \text{ is unmarked, the nonblank entries of state } \xi_j \text{ are included in those of } \xi_m \}
\]
The \( r \) members of \( Q_1 \)
\[
\xi_m = \xi_{j_1}, \xi_{j_2}, \ldots, \xi_{j_r}
\]
are arranged in order of descending numbers of nonblank entries.

**Step 2.** A set \( Q_\alpha \) is constructed. Initially its only member is \( \xi_{j_1} \). For \( k = 2, 3, \ldots, r \) in turn, if the nonblank entries of \( \xi_{j_k} \) are included in the nonblank entries of each member of \( Q_\alpha \), then \( \xi_{j_k} \) is included in \( Q_\alpha \). The states in \( Q_\alpha \) can have their nonblank entries overlaid on those of \( \xi_m \) in the list representation of the T-table. The members of \( Q_\alpha \) are marked and a return is made to step 1 to obtain a further compatible set for overlaying.

The algorithm terminates when all states are marked. Note that for many states \( \xi_m \) the corresponding sets \( Q_\alpha = \{ \xi_{j_\alpha} \} \).

In practice this latter process for compacting lists has been applied first and the other technique has been applied only to those states \( \xi_m \) for which the corresponding set \( Q_\alpha = \{ \xi_{j_\alpha} \} \).

### 5.3. Encoding of the N-Table

The encoding used for the N-table is basically similar to that of the T-table, using three corresponding arrays NSTATE, NSYM and NACT. As blank entries in the N-table can never be encountered by the parsing algorithm, the default entry technique used in encoding the T-table may be applied to eliminate a frequently occurring entry from either a row or a column of the N-table. Elimination of chain derivations ensures that in some rows several entries are identical. In columns, repetitions of the same next
state can be expected, since all next state entries leading to a particular row must lie in the same column. There is little doubt that choosing a default value for each column is superior and is extremely effective in reducing the space requirements. The choice of default entry is a value occurring at least as often as any other in a column. An array NDEF contains the default value for each column of the N-table, and the arrays NSYM and NACT contain only those entries from a row which differ from the default value of the corresponding column.

Looking up an entry in the N-table involves a search through the list in NSYM which is accessed via NSSTATE, and, if an entry matching the given nonterminal symbol is found in NSYM, the corresponding element in NACT is selected, otherwise the nonterminal symbol is used to select an element from NDEF.

Overlapping of entries in the arrays NSYM and NACT is possible; unless chain derivations are eliminated, the overall benefit here is very much less, since the default entry technique results in these arrays being small compared to TSYM and TACT. The elimination of chain derivations results in columns of the N-table becoming identical or in the nonblank entries of one column becoming a subset of those in another column. Both conditions are apparent in figure 6. Basing the list representations on columns rather than rows allows appreciable space saving by overlaying, but the lengths of the lists to be searched tend to be longer than those based on rows for larger grammars. Again, to keep the length of the lists short, the N-table representation figure 6 is preferred to that of figure 7 as a basis for constructing the lists.

6. Implementation and Results

Programs have been constructed to produce SLR(1) state tables and to perform most of the transformations described, thereby enabling the effects of the latter to be examined when applied to grammars for existing languages.

AlgolW is essentially the language described by Wirth and Hoare (1966). It is of interest here for several reasons. The language is reasonably large, containing all of the facilities of Algol60 with several significant extensions. The compiler for the IBM360 constructed at Stanford University is fast, notwithstanding its three passes, and produces efficient code. Its speed is due,
in no small measure, to the parsing method used. The parser is based on the simple precedence method of Wirth and Weber (1966), but it contains a significant (unpublished) extension due to Susan Graham which enables chain derivations to be bypassed when parsing simple precedence languages. It is largely the capability of bypassing chain derivations which accounts for the well known speed of operator precedence methods and the similar ad hoc parsers. The existing parser for the AlgolW compiler is thus a suitable subject with which to make space and time comparisons. The compiler is written in PL360 (Wirth, 1968) making it trivial to ascertain the existing parsing table space requirements and a relatively simple matter to replace the parser for timing comparisons.

The XPL compiler (McKeeman, Horning and Wortman, 1970) uses a mixed strategy precedence parser with one character of lookahead. The class of grammars accomodated by this method is larger than and properly includes the simple precedence grammars. XPL is of interest since Lalonde (1971) has already replaced the mixed strategy precedence parser in the XPL compiler by an LALR(1) parser and reported on the space and time comparisons of his implementation.

6.1. Space Efficiency

To put space requirements in context the formats used for array elements are described briefly. TSYM and NSYM have 8 bit elements permitting a 255 symbol vocabulary. Since the state tables are split into a T-table and an N-table then 255 symbols in each of \( V_n \) and \( V_T \) are in fact permitted). TSTATE and NSTATE have 16 bit elements which specify both a pointer and a length; using 10 bits for the pointer allows approximately 1024 nondefault entries from each of the N-table and the T-table, provided that no more than 64 entries occur in a single row of these tables. The use of 16 bit elements for the arrays TACT, NACT and NDEF reflect convenience of access on an IBM System/360 rather than information content. These elements include a two bit field distinguishing the type of state table entry - next state, production number or scan-production number - and an integer defining either the next state or production number. As space is available within these elements, the length of the RHS of a production is included for production and scan-production entries; these lengths could equally well be stored in a separate array, as indeed the LHS symbols of the productions are. Eight bit entries suffice for the array LHS.
It is perhaps appropriate to interpolate a practical comment on the use of LR analysers. Programming language grammars are not usually strictly context free and so are not LR(k), much less SLR(1). While the lexical analyser and semantic routines of a compiler can usually be so designed that an SLR(1) grammar represents the syntax of a language, it may not always prove convenient or possible. The failure of a grammar to satisfy the SLR(1) condition implies that for one or more state-input symbol combinations there is not a unique action in the T-table. The list of possible alternative actions for one of these state-input combinations can contain at most one action of the next state or scan-production number type; the others are necessarily of the production number type. In the current implementation it is assumed that a semantic routine, uniquely associated with a particular production, is invoked whenever a production number or scan-production number entry naming that production is encountered. Provided that this semantic routine can determine whether or not its associated production may be applied to reduce the stack at this point in the parse, such lists of actions can be accommodated by means of a fourth type of entry in the TACT array, which specifies both the number of alternative entries and points to the first of the alternatives in an array SUPTACT. The entries in SUPTACT have the same format as those in TACT. A single next state entry, if one exists, is placed last in the list of alternative actions and is used only if no production number entry is found to be applicable. The amendment to the parsing routine to deal with this extension involves the inclusion of a flag which is set on encountering an entry of the fourth type in TACT and which is reset either by a semantic routine which determines that its corresponding production may be applied or by the parsing routine if the last entry is of the next state type. The parsing routine selects alternatives from the list passing control to the appropriate semantic routines so long as the flag is set.

Typically, this mechanism may be used to handle the concept of types associated with identifiers in many programming languages.

The space compaction techniques described can usefully be applied to these extensions to the T-table. Two rows of the T-table can be merged if for each input symbol there are identical unique actions or identical lists of nonunique actions. In any row of the T-table, a list of alternatives containing only production
number entries may be chosen as the default entry for the row. In the array SUPTACT lists may be overlaid.

Table 1 provides information on the size of the Algol W and XPL grammars together with details of the SLR(1) state tables for these grammars. The effect on the tables of the various transformations is indicated, and storage requirements are given based on the list representation data structures which have been described. Finally, for comparison, the storage requirements of the existing parse tables in the current Algol W and XPL compilers are included. The table sizes given in table 1 result from using (9) to partially eliminate chain derivations. If no eliminations were attempted, the arrays NSYM and NACT would have contained 86 fewer elements in the case of Algol W and 12 fewer for XPL.

The space requirements for XPL are slightly better than those reported by Lalonde for his implementation. The data structures used here lead to a somewhat simpler parsing routine and even without considering elimination of chain derivations rather more optimisation to reduce parsing time has been employed here.

Table 2 provides similar data, for Algol W only, when chain derivations are completely eliminated. Without the overlaying techniques of section 5.2, the space requirements compared with partial chain elimination would have quadrupled, mainly because the number of entries in TSYM and TACT have increased sevenfold. Lesser contributions arise from the doubling of the number of states to be recorded and the fact that use of default entries for columns of the N-table is less effective in saving space. The overlaying techniques recover much of this extra space so that the final space requirements quoted in table 2 are only about twice those of table 1. It is clear that space requirements for languages with many operators in an extensive hierarchy would be great. On breaking in the middle, the 12 step chain derivation for an Algol W expression (by stipulating the existence of a semantic routine for the appropriate production), then creating parse tables with full chain elimination specified, it was observed that the resulting tables had a space requirement midway between those reported in tables 1 and 2. Time-space trading is possible in this way.

In comparison with the current Algol W parser, the SLR(1) parser fares favourably on space requirements even when full chain elimination is specified.
<table>
<thead>
<tr>
<th></th>
<th>ALGOLW</th>
<th>XPL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Grammar</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Productions</td>
<td>190</td>
<td>108</td>
</tr>
<tr>
<td>Terminal symbols</td>
<td>62</td>
<td>41</td>
</tr>
<tr>
<td>Nonterminal symbols</td>
<td>71</td>
<td>48</td>
</tr>
<tr>
<td>Average length of production RHS.</td>
<td>2.32</td>
<td>2.06</td>
</tr>
<tr>
<td><strong>SLR(1) state tables</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States</td>
<td>330</td>
<td>183</td>
</tr>
<tr>
<td>Entries in terminal columns</td>
<td>4513</td>
<td>1178</td>
</tr>
<tr>
<td>Entries in nonterminal columns</td>
<td>1552</td>
<td>395</td>
</tr>
<tr>
<td>States with nonunique entries</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Nonunique entries</td>
<td>837</td>
<td>0</td>
</tr>
<tr>
<td><strong>Effects of transforming state tables</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LR(0) states eliminated</td>
<td>166</td>
<td>84</td>
</tr>
<tr>
<td>Identical rows eliminated from T-Table</td>
<td>37</td>
<td>23</td>
</tr>
<tr>
<td>Rows with nonunique entries eliminated from T-Table</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Entries eliminated using T-table defaults</td>
<td>2543</td>
<td>731</td>
</tr>
<tr>
<td>Entries eliminated using N-table defaults</td>
<td>1338</td>
<td>327</td>
</tr>
<tr>
<td>Entries eliminated by removing identical T-table rows</td>
<td>526</td>
<td>146</td>
</tr>
<tr>
<td>Entries eliminated by overlaying rows in the T-table list representation</td>
<td>148</td>
<td>64</td>
</tr>
<tr>
<td><strong>SLR(1) tables in list form</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Array elements in:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSTATE</td>
<td>160</td>
<td>99</td>
</tr>
<tr>
<td>NSTATE</td>
<td>60</td>
<td>39</td>
</tr>
<tr>
<td>TSYM,TACT</td>
<td>323</td>
<td>153</td>
</tr>
<tr>
<td>NSYM,NACT</td>
<td>214</td>
<td>68</td>
</tr>
<tr>
<td>LHS</td>
<td>190</td>
<td>108</td>
</tr>
<tr>
<td>NDEF</td>
<td>71</td>
<td>48</td>
</tr>
<tr>
<td>SUPTACT</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

Total storage requirements in bytes for:
- SLR(1) parser with partial chain elimination 2419 1143
- Simple Precedence parser (with complete chain elimination) 6730
- Mixed Strategy Precedence parser (with no chain elimination) 2962

Table 1. Storage space for ALGOLW and XPL parse tables with chain derivations partially eliminated.
SLR(1) state tables

States 469
Entries in terminal columns 7483
Entries in nonterminal columns 1556
States with nonunique entries 7
Nonunique entries 837

Effects of transforming state tables

LR(0) states eliminated 155
Identical rows removed from T-table 37
Rows with nonunique entries eliminated from T-table 4
Entries eliminated using T-table defaults 3455
Entries eliminated using N-table defaults 1022
Entries eliminated by removing identical T-table rows 524
Entries eliminated by overlaying rows in the T-table list representation 1605

State Tables in List Form

Array elements in: TSTATE 314
NSTATE 61
TSYM,TACT 937
NSYM,NACT 534
LHS 190
NDEF 71
SUPTACT 18

Total storage requirement in bytes: 5531
Simple Precedence 6730

Table 2. Storage space for ALGOLW parse tables with full elimination of chain derivations.
<table>
<thead>
<tr>
<th>Program</th>
<th>SLR(1)</th>
<th>SP</th>
<th>SLR(1)PC</th>
<th>SLR(1)SC</th>
<th>SLR(1)C</th>
<th>SFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>0.43</td>
<td>0.41</td>
<td>0.35</td>
<td>0.31</td>
<td>0.28</td>
<td>0.27</td>
</tr>
<tr>
<td>Program 2</td>
<td>1.12</td>
<td>1.09</td>
<td>0.94</td>
<td>0.83</td>
<td>0.76</td>
<td>0.75</td>
</tr>
<tr>
<td>Program 3</td>
<td>1.64</td>
<td>1.59</td>
<td>1.39</td>
<td>1.21</td>
<td>1.09</td>
<td>1.10</td>
</tr>
<tr>
<td>Program 4</td>
<td>2.40</td>
<td>2.32</td>
<td>2.02</td>
<td>1.78</td>
<td>1.61</td>
<td>1.60</td>
</tr>
<tr>
<td>Program 5</td>
<td>4.07</td>
<td>3.92</td>
<td>3.42</td>
<td>3.02</td>
<td>2.71</td>
<td>2.75</td>
</tr>
</tbody>
</table>

Table 3. Time (seconds) of the syntax analysis phase of the AlgolW compiler using various parsing methods.
6.2 Time Efficiency

The lexical and syntax analysis phase of the AlgolW compiler was timed using several AlgolW programs as input and with four variants of the SLR parser substituted for the existing parser. As it was easy to suppress the effects of chain elimination in the latter, timings with and without it were taken for comparison. The performance ranking of the six parsers was substantially the same whichever program was used for input. The results for five of these programs are shown in Table 3; times are quoted in seconds and are not more accurate than ±0.01 seconds.

In order of increasing speed the parsers are:

1) SLR(1). An SLR(1) parser with LR(0) reduce states removed but with no chain elimination attempted.

2) SP. The existing simple precedence parser of the AlgolW compiler but with elimination of chain derivations suppressed.

3) SLR(1)PC. The parser for which space data was given in Table 1. Chains are partially eliminated (to the extent that they can be without increasing the number of states).

4) SLR(1)SC. The parser mentioned at the end of the previous section in which the chain derivation for an AlgolW expression was broken in the middle but otherwise chains were completely eliminated.

5) SLR(1)C. The parser for which space data was given in Table 2 and in which the chain derivations have been completely eliminated.

6) SPC. The existing AlgolW parser.

The most important comparison is between the SLR(1)C and the SPC parser. The slight difference in performance between the two parsers can be viewed as resulting from minor optimisations in the SPC parser; similar optimisations are available within the SLR framework but more direct methods of improving performance of SLR parsers are available and look more promising. For example, after complete elimination of chains, overlaying of columns of the N-table (after default entries have been removed) offers a way of replacing list searching by direct table look up without increasing space requirements enormously.

The fact that performance differences between the SLR(1) and SP parsers are so small merits comment; particularly since Lalonde reported that his LALR(1) parser (using software list searching) produced parsing times which were about 60% of those given by the
mixed strategy precedence parser in the XPL compiler. This
difference is largely accounted for by the different ways in which
the simple precedence and mixed strategy precedence methods determine
which production to use when the stack must be reduced. The
simple precedence method determines the number of symbols which must
be removed from the stack—a fact used by the SP parser in setting
up a hashing function which is known to be extremely efficient
(thereby largely negating the potential advantage of the LR methods,
namely, knowing a priori which production to apply). In the
absence of knowledge of the length of the production, the mixed
strategy precedence method involves a search through all
productions with a final symbol matching that at the top of the
stack. Two further comments may be made. Unpacking two bit
precedence entries is expensive; the SP parser expends 3800 bytes
to avoid this penalty. Secondly, pressing the translate and test
instruction into service is not an ideal way of searching lists,
particularly in view of the abysmal indexing capabilities of the
IBM System/360 character handling instructions. The overhead for
searching short lists is high.

In terms of time space trade-offs, table 3 suggests that the
PC variant (2419 bytes) gives a substantial performance improvement
for the modest space premium (258 bytes) when compared with the
straightforward SLR(1) parser. The SLR(1)SC and SLR(1)C variants
with space requirements of approximately 4000 and 5500 bytes are
less economical.

7. Summary

The parsing methods based on the SLR(1) algorithm accept a wider
class of grammars than other formal methods used at present, with
consequential weakening of constraints in choosing a grammar which
conveniently reflects the semantics of a language. While this
is perhaps the major advantage of these methods, the early detection
of syntactic errors has immediate benefit to both the compiler writer
and the compiler user and has potential benefit in the error recovery
problem.

The implementation described here was deliberately directed
towards exploiting the list searching instructions available on
computers such as the PDP10, Univac 1108 and IBM 360; it was
shown that in such an environment the above advantages can be

32.
obtained with time and space efficiencies quite comparable to one of the better methods in use. The absence of hardware assisted list searching does not necessarily imply untenably large space requirements to achieve high performance. Suggestions for compactly encoding the matrix form of the state tables were made in passing; also since typically there are many states in the state table having only one or two entries, mixed matrix - list representations are possible. If the early error detection requirement is relaxed, further compaction techniques become available which have been excluded here.

Finally given a particular representation of the state tables, the parsing routine which utilises them is independent of whether the tables are SLR(1), LALR(1) or LR(1). Likewise the table compaction techniques are, in the main, applicable to all three kinds of table although they have been described in the context of SLR(1) tables.

Acknowledgements It is a pleasure to acknowledge many discussions with E.H. Satterthwaite and W.C. Lynch. Thanks are also due to the former and J.M. Rushby for reading drafts of this manuscript and pointing out deficiencies.

Appendix

The sets first \((x)\) and follow \((A)\) may be computed from the relations given below.

First nonterminal symbols generating the empty string are determined recursively using

\[
A \xrightarrow{*} \Lambda \text{ iff } A \rightarrow B_1 B_2 \ldots B_n, \ B_i \xrightarrow{*} \Lambda, \ 1 \leq i \leq n, \ n \geq 0.
\]

For a terminal symbol \(x\), \(\text{first } (x) = \{x\}\).

For a nonterminal symbol \(A\) we have

\[
\text{first } (A) \text{ iff } X = A \cup A \rightarrow B_1 \ldots B_n Y, \ B_i \xrightarrow{*} \Lambda, \ 1 \leq i \leq n, \ n \geq 0, \ X \in \text{first } (Y)
\]

If \(\alpha \in \V^*\) then

\[
X \in \text{first } (\alpha) \text{ iff } \alpha = Y\beta, \ X \in \text{first } (Y) \cup (Y\delta \cup \Lambda X \in \text{first } (\beta))
\]

The sets last \((x)\) are needed in the computation of follow \((A)\) and are obtained analogously to first \((x)\), i.e.,

\[
\text{last } (x) = \{x\}
\]

33.
X ∈ last (A) iff \( X = A \uparrow \)
\[ A \rightarrow \text{BY}_1 \ldots \text{B}_n, \text{B}_i \rightarrow \lambda, 1 \leq i \leq n, n \geq 0, X \notin \text{first} (X) \]

Finally
\[ X \in \text{follow} (Y) \iff \exists A \rightarrow \text{BY}_1 \ldots \text{B}_n \uparrow \gamma, \text{B}_i \rightarrow \lambda, 1 \leq i \leq n, n \geq 0, \\
Y \in \text{last} (Y_i), X \in \text{first} (X_i) \]

References

Department of Electrical Engineering, M.I.T.


Feldman J.R. and Gries D., 'Translator Writing Systems', CACM 11 (1968) 77


Gries D., 'Use of Transition Matrices in Compiling', CACM 11 (1968) 26

Knuth D.E., 'On the Translation of Languages from Left to Right', Information and Control 8 (1965) 607


Lewis P.M. and Stearns R.E., 'Syntax Directed Transduction', JACM 15 (1968) 465
McKeeman W.M., Horning J.J. and Wortman D.B.,

'A Compiler Generator' (1970)
Prentice Hall

Pager D.,

'A Solution to an Open Problem by Knuth'
Information and Control 14 (1970) 462

Wirth N.,

'PL360, A Programming Language for the 360 Computers',
JACM 12 (1968) 37

Wirth N. and Hoare C.A.R.,

'A Contribution to the Development of Algol',
CACM 9 (1966) 413

Wirth N. and Weber H.,

'EULER: A Generalisation of Algol, and its Formal Definition, Part I',
CACM 9 (1966) 13